MANIP — A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems

BENJAMIN W. WAH, MEMBER, IEEE, AND Y. W. EVA MA, MEMBER, IEEE

Abstract — **In this paper, we propose and analyze the design of MANIP, a parallel machine for processing nondeterministic polynomial (NP)-hard problems. The most general technique that can be used to solve a wide variety of NP-hard problems on a uniprocessor system, optimally or suboptimally, is the branch-and-bound algorithm. We have adapted and extended branch-and-bound algorithms for parallel processing. The parallel branch-and-bound algorithm requires a combination of sorting and merging that will be too inefficient to perform in a common memory. We have proposed a system with distributed intelligence so that sorting can be carried out efficiently. A unidirectional ring network is shown to be the most cost-effective interprocessor communication network. The performance on the proposed system is evaluated using the vertex-covering problem.**

Index Terms — **Approximate branch-and-bound, NP-hard problems, parallel branch-and-bound, ring network, selection, vertex-covering problem, virtual memory.**

I. INTRODUCTION

THE class of *NP-hard* problems [6] is very important and spans a wide spectrum of application areas. Various problems like traveling salesman, warehouse location, job-shop scheduling, graph partitioning, dynamic storage allocation, consistency of database frequency tables, integer programming, and feasible register allocation for program optimization have been proven to be NP-hard.

An important characteristic of NP-hard problems is that there is no known optimal algorithm to solve the problems with a computation time that increases polynomially with the size of the problem. The computation time for all known optimal algorithms for this class of problems increases at least exponentially with the problem size. That is, if n represents the size of the problem, the computation time goes up at least as fast as k^n with $k > 1$.

To illustrate the extent of this complexity, suppose a given computer can solve a problem of size 10 in 0.001 s. If the problem has a time complexity of 2^n, then the same computer can solve a problem of size 60 in 357 centuries. Moreover, improved technologies appear to have little effect on improving the problem size that can be solved. For example, the computer just described can solve a 2^n problem with $n = 31$ in 1 h. With a computer 1000 times faster, the problem size that can be solved is 41. With a computer 10 000 times faster, the problem size that can be solved is 45.

It is obvious that improving technologies alone is not enough to expand the solvable problem space of NP-hard problems. Likewise, parallel processing by itself does not provide a solution. A general result concerning the exponential nature of NP-hard problems is that the product of the degree of parallelism and processing time is exponential. To achieve an exponential reduction in processing time, an exponential number of processors must be used. This has been proven for the knapsack problem [24]. Parallel processing is, thus, most rewarding when the problem is NP-hard, but is solvable in polynomial time on the average [29], or the problem is heuristically or approximately solvable in polynomial time.

In practice, approximate solutions are acceptable alternatives. The simplest approach is to solve small problems optimally, and to apply heuristics to solve large problems suboptimally. The drawback of this approach is that it is difficult to assess the goodness of results. Polynomial-time approximation algorithms with guaranteed bounds have been developed, but often they are problem-dependent and are devised with mathematical ingenuity [9], [18]. It is preferable to use a general algorithm that can solve, optimally or suboptimally (with guaranteed bounds), a wide variety of NP-hard problems. A good candidate is the branch-and-bound algorithm.

A combination of two techniques are proposed in this paper for evaluating NP-hard problems. The generality and approximation behavior of branch-and-bound algorithms allow them to be applied to a wide variety of NP-hard problems. Although the complexity is exponential inherently, it has been shown that an exponential reduction in iterations can be achieved with a linear reduction in accuracy [22]. Parallelism, on the other hand, can be used to linearly reduce the number of iterations. Approximation and parallelism can be applied together, and they represent feasible approaches to reducing the complexity of branch-and-bound algorithms.

The goal of this paper is to study a parallel version of the branch-and-bound algorithm that can be executed efficiently on a multiple computer system. Conventionally, branch-and-bound algorithms have been studied with respect to limited memory space, selection and bounding criteria, theoretical behavior, and adaptation to a single computer system. Very few studies have been done on the design of a parallel computer architecture and its associated operating system which can provide an execution environment for branch-and-

Manuscript received April 7, 1983; revised August 19, 1983. This work was supported by National Science Foundation Grant ECS81-05968.

B. W. Wah is with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

Y. W. E. Ma is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

0018-9340/84/0500-0377$01.00

bound algorithms. We will concentrate on the problem of architectural design in this paper. The computational efficiency and virtual-memory support are presented elsewhere [22], [28], [30].

Many studies have been made to design parallel algorithms in order to speed up solutions of search problems. Examples include graph traversal, minimum spanning tree, shortest path, graph biconnected component, maximum flow, alpha–beta search [13], and decision tree evaluation [12]. A variety of SIMD and MIMD interconnection networks have been proposed for processor–processor communication or processor–memory communication [5]. However, these networks are designed for general purpose applications, and hence are not tailored to the special features of processing NP-hard problems. Increasing the number of processors usually yields rapidly diminishing returns.

Several architectures have been proposed to solve NP-hard problems [3]. All of them used implicit enumeration to find the optimal solution. Basically, the system dedicates one subproblem to each processor, and this processor reports to its parent processor when the evaluation is completed. Because of the limited degree of communications, some processors might be working on tasks that could be eliminated if a better interconnection network was designed. In general, implicit enumeration is time-consuming and wasteful, and for NP-hard problems, the critical issues of exponential space or exponential time must be addressed in the algorithm.

Research on applying branch-and-bound algorithms to solve NP-hard problems in a network of computers was done by Imai *et al.* [26] and El-Dessouki and Huen [4]. The algorithm is based on a general-purpose network architecture with limited memory space and slow interprocessor communication. Due to memory limitations, a depth-first search criteria (to be discussed later) is used. This is not very effective in minimizing execution time. With the availability of VLSI technology, larger and inexpensive memories, and faster communication technology, reducing the execution time (at the expense of larger memory space requirement and more interprocessor communication) becomes a very important problem.

In this paper, we propose the design of MANIP, an architecture implementing parallel branch-and-bound algorithms with a best-first search (to be discussed later). Our study identifies the necessary architectural features, and therefore would provide insights to evaluating NP-hard problems on general parallel processing systems. The critical element in this architecture is the selection network which selects subproblems to be evaluated by the parallel processors. This is investigated in this paper. Another problem faced in the implementation is the limited main memory size. With larger and inexpensive secondary storage, branch-and-bound algorithms can be designed with a virtual backing store. The conventional virtual memory system does not work very efficiently here because the access characteristics of branch-and-bound algorithms are significantly different from the access characteristics of general programs. A modified branch-and-bound algorithm that can overlap all the secondary storage accesses with computations is discussed elsewhere [22], [25], [30]. A third problem faced in the implementation is the existence of anomalies with parallel processing [27]. In [28], [30], conditions leading to anomalies and the appropriate measures were investigated.

The system was designed according to the following design objectives:

1) it should be modularly expandable to include a very large number of processors;

2) the design must have high performance, and the cost should be kept low by replicating simple cells;

3) it should use distributed control so that a controller would not become the bottleneck in future system expansion;

4) efficient load balancing strategies should be implemented;

5) it should be recoverable from hardware failures.

In the following sections, the branch-and-bound algorithm and its parallel version are presented. The architecture to implement parallel branch-and-bound algorithms is identified. The design tradeoffs and an evaluation of the selection network are presented. Finally, the system is evaluated by using the vertex-covering problem.

II. Branch-and-Bound Algorithms

Branch-and-bound algorithms have been extensively studied in areas such as artificial intelligence and operations research, and have been applied to solve NP-hard problems. The type of problems solvable by a branch-and-bound algorithm can be put into a constrained optimization

$$\begin{aligned} &\text{Minimize} && C_O(x) \\ &\text{Subject to} && g_1(x) \geq 0 \\ &&& g_2(x) \geq 0 \\ &&& \cdot \\ &&& \cdot \\ &&& \cdot \\ &&& g_m(x) \geq 0 \\ &\text{and} && x \in X \end{aligned}$$

in which X represents the domain of optimization defined by the m constraints, normally a Euclidean n-space, and x denotes a vector $(x_1, x_2, \cdots, x_n)$. Problems expressed in this form may or may not be NP-hard. A solution vector x that lies in X is called a *feasible solution*, and a feasible solution for which $C_O(x)$ is minimal is called an *optimal solution*.

A. Essential Features of Branch-and-Bound Algorithms

A branch-and-bound algorithm [11], [14] is an organized and intelligently structured search of the space of all feasible solutions. Generally, the space of all feasible solutions is repeatedly partitioned into smaller and smaller subsets, and both the lower and upper bounds are calculated for values of solutions within each subset. After each partitioning, subsets with lower bounds (in the case of minimization) that exceed either the value of a known feasible solution or the least upper bound of all subsets are excluded from further partitioning. The partitioning process continues until a feasible solution is found such that the value is no greater than the lower bound of any subset.

The state of the partitioning process at any time can be represented as a partial tree. Each terminal node in the tree

represents a partition and is called a *subproblem*. The partitioning process selects a partition and breaks up this partition into smaller partitions. This extends the node in the partial tree representing this partition by one level and uses the sons to denote the smaller partitions. Each leaf node has two numbers associated with it: the upper bound and the lower bound of the subproblem. A leaf node whose lower bound is less than both the value of all known feasible solutions and the least upper bound of all leaf nodes is *active;* otherwise, it is *terminated* and need not be considered in any further computation.

There are two essential features of a branch-and-bound algorithm: the branching rule and the bounding rule. The branching rule examines the set of active leaf nodes and, based on some predefined criterion, selects one for expansion. If the set of active nodes is maintained in a first-in/first-out list, the algorithm is called a *breadth-first search*. If the set is maintained in a last-in/first-out list, the algorithm is called a *depth-first search*. Lastly, if the node selected for expansion is one with a minimum lower bound, the search algorithm is called a *best-first search*.

In a breadth-first search, the nodes of the tree will always be examined in levels; that is, a node at a lower level will always be examined before a node at a higher level. This search will always find a goal node nearest to the root; however, the sequence of nodes examined is always predetermined, so the search is "blind." The depth-first search has a similar blind behavior except that a subtree is generated completely before other subtrees are examined. In both algorithms, the next node to be examined is known. The state of the parent nodes do not have to be kept and can be regenerated each time when the parent nodes are referenced. These two algorithms, therefore, are space saving.

In contrast, the best-first search is space consuming because all active subproblems must be stored as intermediate data in the computer. The total number of nodes expanded, however, is minimized in the sense that any node expanded under a serial best-first search must also be expanded under other search policies, provided that all the bounds are unique [11]. The reason for this stems from the fact that only subproblems with lower bounds smaller than or equal to the optimal solution will be evaluated.

Once the subproblem has been selected for partitioning, some undetermined parameters in the subproblem must be selected so that alternatives for these parameters can be defined and multiple subproblems can be created. For example, in the traveling-salesman problem [6], the undetermined alternatives are the set of untraversed edges. In expanding a subproblem, an untraversed edge (i, j) is selected and two alternatives can be created: 1) the edge is traversed and the salesman goes directly from city i to city j; and 2) the edge is not traversed. The parameter for expansion is usually chosen in an ad hoc manner.

After new subproblems are created, the bounding algorithm is applied to evaluate the upper and lower bounds of a subproblem. Generally, only the lower bound is evaluated because the merits of using the upper bound are small. The bounding algorithm that is designed is highly dependent on the problem. For example, in an integer-programming problem, a linear program with relaxed integer constraints can be used as a lower bound [14]; in the traveling-salesman problem, an assignment algorithm or a spanning-tree algorithm [8] can be used as the bounding algorithm.

The vertex-covering problem [6] which is used as the basis of performance evaluation in our study is an example of an NP-hard problem. In this problem, the minimum number of vertices that are needed to "cover" all the edges in an undirected graph is to be found. (Cover means that all the edges in the graph emanate from at least one of the included vertices.) The branching rule uses the best-first search and branches on an unselected vertex with the largest out-degree, from which two subproblems can be created: one including the vertex in the set and one excluding it. In calculating the lower bound, the out-degrees of the edges are based on the uncovered edges and not on the edges of the original graph. The lower bound in the bounding rule is chosen to be the minimum number of unselected vertices such that the total out-degree of these vertices is greater than or equal to the number of uncovered edges. Note that edges emanating from different vertices in the lower-bound calculation may overlap, and therefore this lower bound does not necessarily cover all the uncovered edges. Further, if a vertex has been excluded in a previous step, and there are uncovered edges emanating from this excluded vertex in the current subproblem, the unselected vertices covering these edges must be included in the minimal set first before the lower bound is calculated. As an example, the branch-and-bound tree for the graph in Fig. 1(a) is shown in Fig. 1(b). For example, in the left subtree of the root [Fig. 1(b)], node 4 of the graph is included. There are six remaining edges. If nodes 1 and 3 are included, the sum of the out-degrees is six. Therefore, the lower bound is three nodes. Similarly, in the right subtree of the root, node 4 is excluded. Thus, nodes 2, 5, 6, and 7 must be included in the minimal set first. There is one edge, (1, 3), that remains uncovered. The out-degrees of nodes 1 and 3 are both one (the covered edges are eliminated). It takes one node to cover edge (1, 3). The lower bound is, therefore, five nodes.

B. Parallel Branch-and-Bound Algorithms

Four sources of parallelism are identified in a branch-and-bound algorithm: parallel evaluation of subproblems, parallel sorting of subproblems, parallel termination tests, and parallel execution of bounding algorithms.

1) Parallel Evaluation of Subproblems: Because multiple subproblems are available, they can be evaluated simultaneously. Owing to overheads in interprocessor communication and sorting, and because some subproblem evaluations are unnecessary, improvement in execution time is usually less than n times (n is the number of processors). For example, Fig. 1(c) shows the parallel evaluation of the branch-and-bound algorithm on the graph in Fig. 1(a) using two processors. It takes three steps in the parallel case as compared to five steps in the sequential case. It was found that a parallel best-first search is more robust with increasing degrees of parallelism as compared to parallel depth-first or breadth-first searches [28]. Since time is critical in evaluating large NP-hard problems, MANIP is designed with the best-

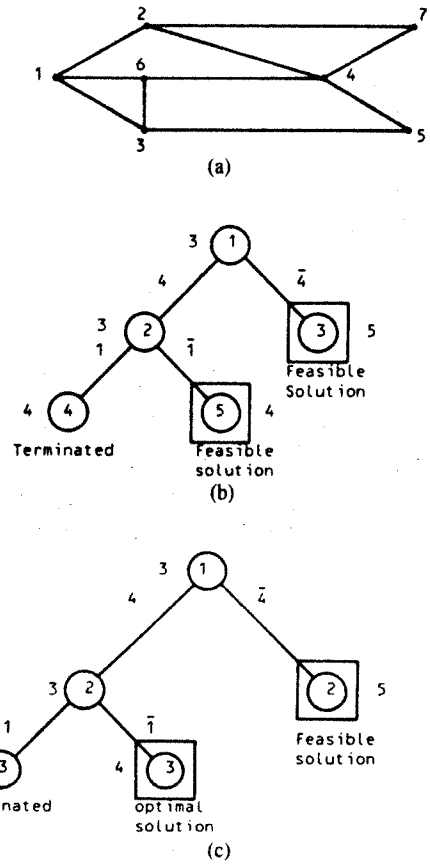


Fig. 1. Example to illustrate the branch-and-bound algorithm. For a branch-and-bound tree: number in node = evaluation order; number outside node = lower bound; number on edge = included or excluded graph vertex. (a) Example graph; (b) branch-and-bound tree for (a); (c) parallel branch-and-bound tree for (a) with two processors.

first branching rule. The large intermediate storage problem can be handled by a virtual memory operating system.

2) Parallel Selection of Subproblems: In a parallel branch-and-bound algorithm using best-first search, a set of subproblems equal in size to the number of processors must be selected in each iteration. Provided that these subproblems are those with the minimum lower bounds, they do not have to be selected in a sorted order. Selection should be carried out in parallel and, preferably, in a time that is less than a linear function of the number of processors. Networks for parallel selection are discussed later.

3) Parallel Termination Tests: Multiple infeasible nodes can be eliminated in each iteration. Further, multiple feasible solutions may be generated, and the minimum value of the feasible solutions has to be updated in parallel.

4) Parallel Execution of the Bounding Algorithm: The evaluation of the bounding algorithm can be sped up by parallel processing. However, when the system is designed for solving general NP-hard problems, the bounding algorithm may be problem dependent. In this case, software implementation of the algorithm is more cost effective.

C. Approximate Branch-and-Bound Algorithms

Branch-and-bound algorithms are useful as general purpose heuristics for computing solutions that differ from the optimum by no more than a prescribed amount [11]. Suppose it was decided at the outset that a deviation of 10 percent from obtained, all subproblems with lower bounds of 136.4 (or 150/1.1) or more will be terminated. This technique significantly reduces the amount of intermediate storage and the time needed to arrive at a suboptimal solution. Techniques are also available to find the best solution in a given length of time. These basically consist of searching for an optimal solution in a length of time equal to $T/2$. If one is not found, then the search is continued for a suboptimal solution that differs from the optimum by no more than half of the maximum error tolerance (5 percent) in a time of length $T/4$. The time for searching is halved each time while the precision of the solution is reduced until a solution is found. All these can be incorporated into the parallel branch-and-bound algorithm.

Little can be said about the efficiency of approximate parallel branch-and-bound algorithms. Some bounds on the efficiency of parallel branch-and-bound algorithms have been developed [7], but these bounds are based on simplistic assumptions and are very loose. For a parallel branch-and-bound algorithm with best-first search, the improvement in the number of iterations will be n times (n is the number of processors) if the number of subproblems in the subproblem list with lower bounds less than or equal to the optimal solution is always greater than or equal to n in each iteration. This improvement is highly problem dependent, but it usually approaches n when the problem is large [28]. Simulation results in Section VII show that the improvement for vertex-covering problems indeed approaches n for large problems.

III. Architecture of MANIP

In this section, we present the general architecture of MANIP. The architectural components that are illustrated can be implemented with hardware, software, or a combination of both.

Fig. 2 depicts the architecture of MANIP. It consists of six major components: selection network, secondary storage redistribution network, secondary storage, processors, global data register, and subproblem memory controllers.

1) Selection Network: The functions of the selection network are

- to select subproblems with the minimum lower bounds;
- to connect the memory controllers so that the workload can be balanced.

The subproblems selected do not have to appear in a sorted order. The design of the selection network is discussed in Sections IV and V.

2) Secondary Storage: Secondary storage provides virtual-memory support of excess subproblems which cannot be stored in the subproblem memory controllers due to memory limitation. Subproblems in a secondary storage module are maintained in a sorted order by the lower bounds. Research on the virtual-memory support is shown elsewhere [25], [30].

3) Secondary Storage Redistribution Network: This network allows a subproblem memory controller to store and access subproblems in any set or subset of the secondary storage modules. Subproblems in the virtual memory can

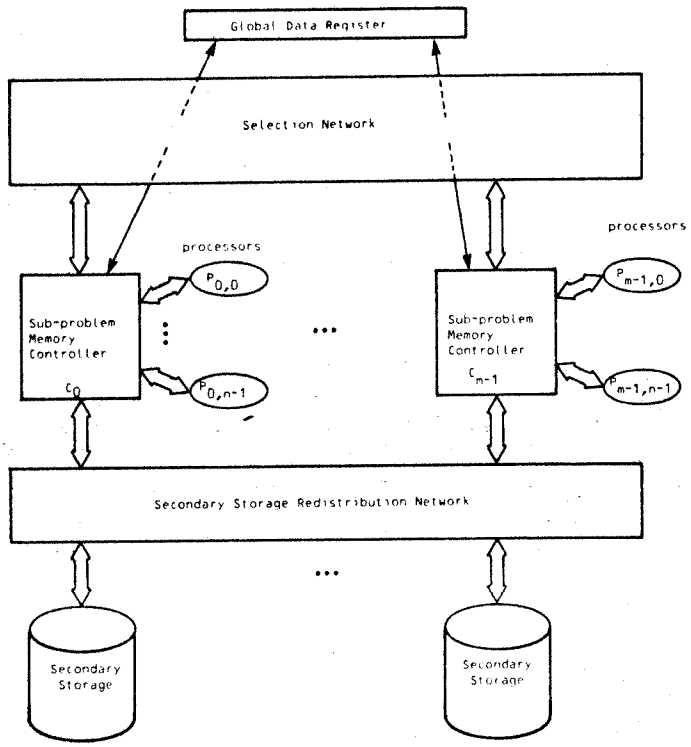


Fig. 2. Architecture of MANIP.

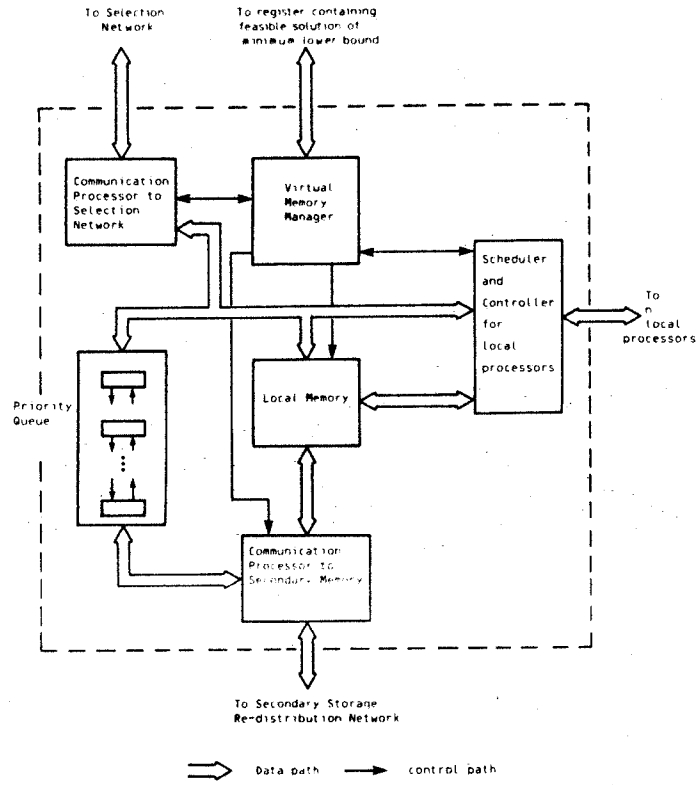


Fig. 3. Architecture of subproblem memory controller.

4) Processors: Processors are general purpose computers for partitioning subproblems and evaluating lower bounds. Because the subproblem size varies for different NP-hard problems, the ratio of the communication time involved to send subproblems to processors and the execution time needed to evaluate lower bounds is not constant. It would be more efficient to allocate a variable number of processors to each memory controller depending on the problem and to change the allocation when needed. This is not done because the network is expensive while the processors are not. Further, the subproblem memory controllers and the processors, although shown architecturally distinct, can be implemented in software in a single computer. Each processor is a process in the system, and a variable number of processes are forked depending on the problem. The optimization on the number of processors required in a problem is studied in Section VI.

5) Global Data Register: This is a register accessible to all memory controllers and contains the minimum value of the feasible solutions. Recall that this value is useful for pruning suboptimal subproblems. When a processor generates a feasible solution, its value is compared to the global minimum value and replaces the global value if it is smaller. Contention may occur at this register if multiple processors try to update this register. A good way to resolve the contention is to use a sequential associative memory [16]. The values of the feasible solutions are shifted out bit-serially and synchronously from all processors into a sequential associative memory that finds the minimum. Because the associative memory is sequential, only m associative memory cells have to be used where m is the number of memory controllers. An associative memory cell capable of minimum searches can be implemented with two flip-flops and four NAND gates.

6) Subproblem Memory Controllers: These controllers tain the virtual-memory operating system, and communicate with other controllers through the selection network. The following functional components are identified (Fig. 3).

- *Priority queue:* The priority queue maintains the local subproblems in ascending order of lower bounds. It is a two-input, two-output deque so that elements can be input or output from the top or bottom. Each element contains a lower-bound value and an address that points to the corresponding subproblem in the local memory. If the structure is implemented in hardware, the elements are stored in registers and comparators are inserted between consecutive elements [10]. For any two consecutive elements, if the top element is greater than the bottom element, these two elements are exchanged. In this way, larger elements "drop" to the bottom of the deque, and smaller elements "float" to the top. Further, elements can be inserted into the deque continuously without waiting for the previous element to be sorted. The structure can also be implemented in software.
- *Communication processors:* The communication processors are responsible for transmitting and receiving subproblems to and from other memory controllers through the selection network, and the secondary storage through the redistribution network. When a subproblem is received by the communication processor, it is written to the local memory. A tag, consisting of the lower bound and the subproblem address in local memory, is written to the priority queue. When the priority queue or local memory is full, the excess subproblems are written to the secondary storage.
- *Local processor scheduler:* The processor scheduler selects subproblems with the minimum lower bounds from the priority queue, picks up these subproblems in the local memory, and sends them to the local processors. After pro-

priority queue and local memory.

• *Virtual-memory manager and local memory:* The memory manager coordinates all the activities in the memory controller. In particular, it manages the pool of local subproblems, and decides on the time to stage subproblems to and from the secondary storage. It selects the secondary storage module to access and is responsible for inserting the subproblems in the proper order in the secondary storage. Another important function of this is to update the global data register and to prune local subproblems with large lower bounds.

The efficiency of the system depends heavily on the capabilities of the selection and redistribution networks. These networks select the right subproblems to expand, and redistribute subproblems so that the workload is balanced. They are discussed in the next section.

IV. Networks for Selection

The (t, k) *selection problem* entails the search of the t smallest numbers from a set of k numbers. This is closely related to sorting except that the t numbers selected do not have to appear in any order. To select $m \cdot n$ subproblems with the minimum lower bounds on a system with m subproblem memory controllers and n processors on each controller, $m \cdot n$ subproblems with the minimum lower bounds are selected from each controller, followed by a selection of $m \cdot n$ subproblems from the $m^2 \cdot n$ subproblems. Since subproblems in each memory controller are sorted, the major problem is the second selection, namely, the $(m \cdot n, m^2 \cdot n)$ selection problem.

Sequential selection can be performed in time $O(m^2 \cdot n)$, but this is inadequate for utilizing the parallelism in subproblem evaluation. This implies that selection must be done in parallel. There are basically two approaches: the use of multistage and of single-stage networks.

B. Multistage Selection Networks

A simple way to perform selection is to sort the numbers first. Parallel sorting networks have been investigated extensively. Batcher pioneered parallel sorting of k numbers by designing a network with $O(k \cdot \log_2^2 k)$ comparators and $O(\log_2^2 k)$ delay [2]. Preparata proposed the fastest parallel sorting algorithm that requires $(c'/\alpha) \log_2 k + o(\log_2 k)$ time and $k^{1+\alpha}$ hardware complexity [15]. However, his design requires more complicated processors and assumes that the k numbers are accessed in parallel without any memory conflict.

In contrast, networks for selection have been little studied. Alekseyev derived the lower $((k-t)\lceil \log_2(t+1) \rceil)$ and upper $((s-1)(2\hat{S}(t)+t)$ bounds on the number of comparators for selecting t elements out of $k = s \cdot t$ elements where $\hat{S}(t)$ is the minimum number of comparators needed for sorting t elements [1]. An effective procedure is based on the selection of t elements out of $2t$ elements. By first sorting $\langle x_1, \cdots, x_t \rangle$ and $\langle x_{t+1}, \cdots, x_{2t} \rangle$, it results in two bitonic sequences. From the properties of bitonic sequences, the t smallest elements can be obtained by comparing and interchanging $x_1 : x_{2t}, x_2 : x_{2t-1}, \cdots, x_t : x_{t+1}$ [2]. To select t elements out of $k = s \cdot t$ elements, the procedure is repeated $s - 1$ times (eliminating t elements each time).

We have improved the above procedure by carrying out the sorting and the $(s-1)$ bitonic merging in parallel [21]. Using Batcher's sorting algorithm, it results in $O(\lceil \log_2 k \rceil \cdot \lceil \log_2 t \rceil)$ time complexity and $O(k \lceil \log_2^2 t \rceil)$ hardware complexity.

Another design using binary comparators was proposed by Yao [23]. He found that for $t < \sqrt{k}$, the number of comparators is $O(k\lceil \log_2(t+1) \rceil)$ and the delay is $O(\log_2 k)$. Networks were proposed which can satisfy either of these complexities, but not both simultaneously. Furthermore, the proposed networks have worse behavior than pure sorting when t approaches $k/2$.

A special property of our $(m \cdot n, m^2 n)$ selection problem is that the $m \cdot n$ lower bound values from each memory controller are already sorted. Therefore, the subsequences can be merged in pairs to form $\lfloor m/2 \rfloor$ subsequences. The larger $m \cdot n$ elements in each of these subsequences can be eliminated and the merging procedure repeats until a single sequence of $m \cdot n$ elements results. Since each subsequence is bitonic, bitonic merging [2] can be applied which results in $O(m^2 n \cdot \lceil \log_2(mn) \rceil)$ hardware complexity and $O(\lfloor \log_2 m \rfloor \lceil \log_2(mn) \rceil)$ time complexity [21]. Another fast merging procedure proposed by Valiant results in $O(m^2 n)$ processors and $O(\lfloor \log_2 m \rfloor \lceil \log_2 \log_2(mn) \rceil)$ time complexity [19]. However, the processors are more complicated, and it is assumed that parallel accesses can be made with no memory conflict.

One important disadvantage of the multistage approach is that the $m \cdot n$ lower bound values from each memory controller must be sent in parallel to the selection network, or they must be accessible in parallel without any memory conflict. This poses a difficult problem for implementation. The single-stage approach proposed next can be implemented easily although it takes a much longer delay for a complete selection. The success of the single-stage approach lies in the fact that it is unnecessary to have a complete selection in order for the parallel branch-and-bound algorithm to work.

B. Single-Stage Selection Networks

The single-stage selection network allows subproblems with the minimum lower bounds in each memory controller to be sent in parallel to k $(k < m)$ neighboring controllers (Fig. 4). The subproblems are then inserted into the local list, and the selection process repeats until m of the $m \cdot n$ subproblems with the minimum lower bounds are distributed to each controller.

All the memory controllers carry out the same distribution operation in each iteration. This mode of operation has the *state preserving property*: that is, if the global minima have been distributed to the controllers, continued redistribution would not disturb the state, and the global minima would remain distributed to the controllers. If different distribution operations are allowed in the memory controllers, the state preserving property may not be achievable. For this reason, we chose to only investigate the first type.

The interconnection network design ranges from a simple unidirectional ring network in which each controller can

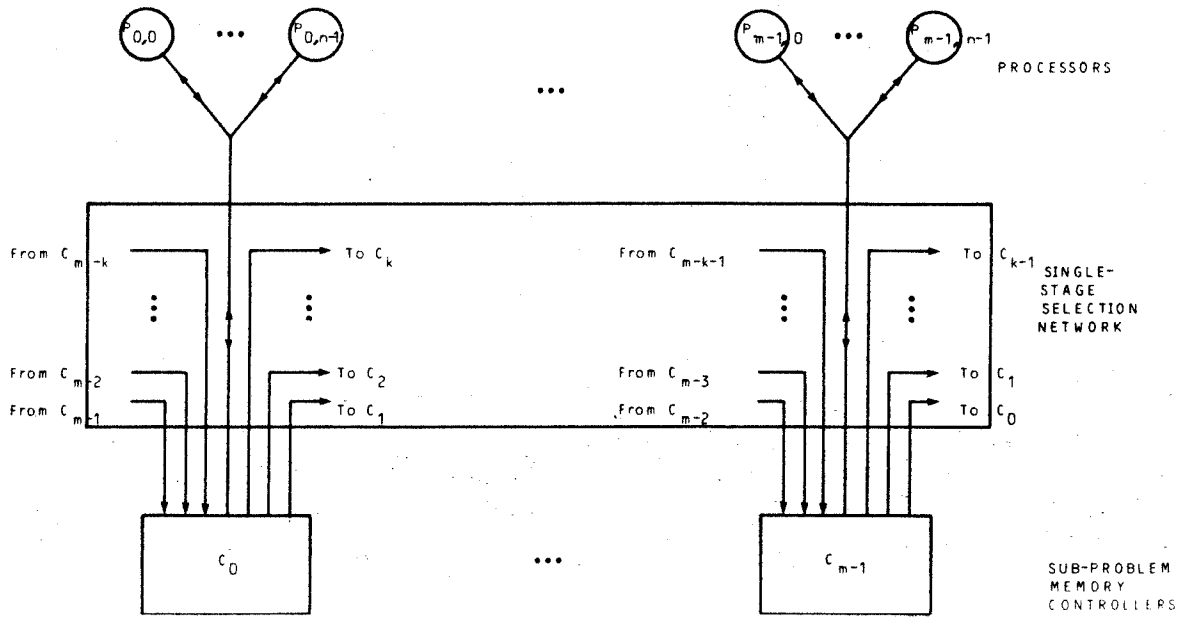


Fig. 4. k-connected single-stage selection network with k subproblems distributed in parallel to k neighboring controllers ($k < m$).

communicate with one of its neighbors, to a fully connected network in which communication can be carried out simultaneously with all the controllers. An analysis in the next section using the urn model shows that a simple unidirectional ring network is adequate.

V. Urn Model — Analysis of Single-Stage Selection Network

The m controllers in the system are represented as m urns that contain $m \cdot n$ white marbles, which stand for the global minima, and $S - m \cdot n$ yellow marbles in which S is the total number of active subproblems. The white marbles are originally distributed randomly to the urns. They are "lighter" than the yellow marbles, so they always "float" to the top of the urn. During the distribution process, one or more marbles are taken from each urn and distributed to one or more other urns in the system. If a white marble exists in the urn, it is always distributed first. The objective is to have a *complete distribution*, that is, each urn has exactly n white marbles.

The ordering of the yellow and white marbles in the urns models the ordering of the subproblems in the controllers. If one or more of the $m \cdot n$ global minima (white marbles) exist in a controller (urn), they are always ordered before the other subproblems (yellow marbles) and are always distributed first. Note that this model does not take into account the ordering of the white marbles, which is important in a conventional sorting and merging problem. It is sufficient for exactly n white marbles to be distributed to each urn regardless of the final order. In the sequel, "marble" is used to represent "white marble."

Distribution strategies refer to the ways that marbles are shuffled in the urns until distribution is completed. Let k = degree of connection; and x = maximum number of marbles that can be received from another urn. In a general strategy, each urn sends out a maximum of $k \cdot x$ marbles to k neighboring urns in parallel. This corresponds to a right cyclic shift on a k-connected ring network. For the

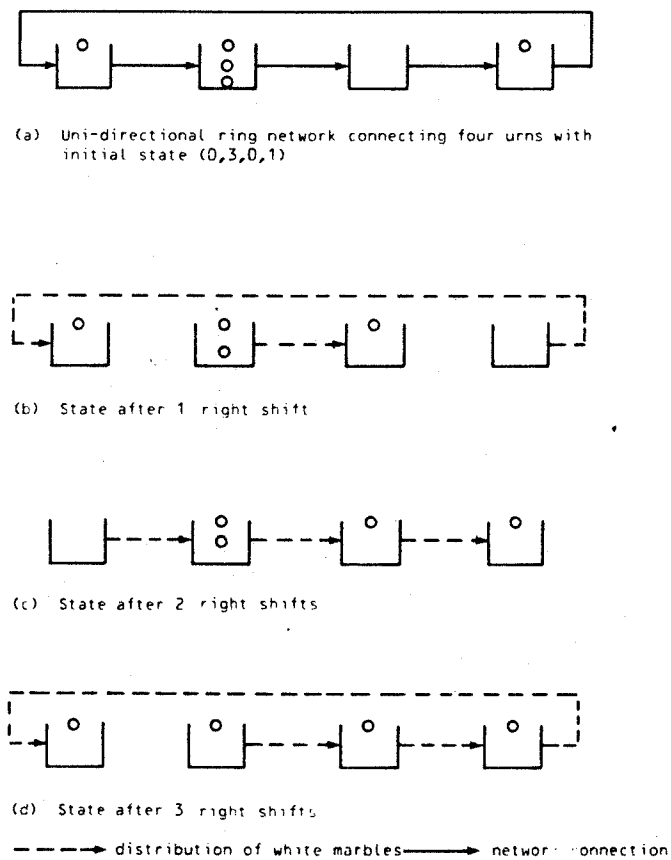
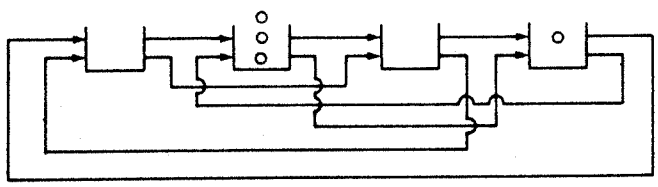


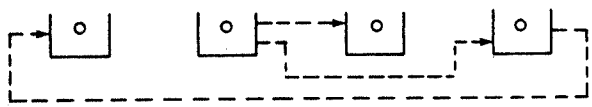
Fig. 5. Unidirectional ring network connecting four urns with $m = 4, n = 1, x = 1$ (only the distribution of white marbles is shown).

ith urn, the $[jx + 1]$st, $\cdots$, $[(j + 1)x]$th marbles (where $j = 0, \cdots, k - 1$) are sent to the $[(i + j + 1) \bmod m]$th urn. When $k = 1$, this becomes a unidirectional ring network. Figs. 5 and 6 show the states of the system after a number of distributions for 1- and 2-connected networks. Only the distribution of white marbles are indicated in the figures.

The parameters k and x must be chosen properly in order



(a) 2-connected network connecting four urns with initial state (0,3,0,1)



(b) State after 1 shift.

⟶ distribution of white marbles

Fig. 6. 2-connected network connecting four urns with $m = 4, n = 1$, $k = 2, k = 1, x = 1$ (only the distribution of white marbles is shown).

to result in a complete distribution. The following theorem relates k and x with n.

Theorem 1: The following properties are necessary and sufficient conditions for a complete distribution.

i) The maximum number of marbles received from another urn must be a factor of the number of marbles to be distributed to each urn ($gcd(x, n) = x$).

ii) The maximum total number of marbles sent out from an urn in each shift must be greater than or equal to the number of marbles to be distributed to each urn if the degree of connectivity is greater than or equal to one ($kx \geq n$ for $k \geq 1$).

iii) If $k = 1, x = n$.

Proof: The proof to show the necessity of these properties is by contradiction.

i) We first prove that $x \leq n$ and then show that x is a factor of n. If $x > n$ and suppose one of the urns has x marbles, these will be sent to a neighboring urn. It will be impossible to break up these x marbles into small subsets because they are always passed as a unit. Therefore, complete distribution is impossible. On the other hand, if $x \leq n$ and x is not a factor of n, then the $m \cdot n$ marbles can be divided into $\lfloor m \cdot n/x \rfloor$ groups with x marbles in each group, and one group with $(m \cdot n - x \cdot \lfloor m \cdot n/x \rfloor) < x$ marbles. These $\lceil m \cdot n/x \rceil$ groups of marbles are distributed randomly to the urns. When a distribution is performed, each group is passed as a unit to a neighboring urn. Continued distribution cannot break up any of these groups. Complete distribution is therefore impossible.

ii) For $k \geq 1$, the condition that $kx \geq n$ is essential. For if $kx < n$, we can assume an initial distribution of kx marbles in each of the $m - 1$ urns and $[m \cdot n - (m - 1)kx]$ marbles in one urn. The extra $[m \cdot n - m \cdot kx]$ marbles in the last urn can never be distributed.

iii) When $k = 1$, we obtain from condition ii) that $x \geq n$, and from condition i) that $x \leq n$, this implies $x = n$.

The proof to show the sufficiency is also by contradiction. Suppose a complete distribution is impossible, there must exist an urn with more than n marbles in each iteration. (The urn that contains more than n marbles may be different in each iteration.) The reasons that the size of this urn does not decrease are either: a) not enough marbles are passed out from an urn each time ($kx < n$); b) too many marbles are passed out from an urn each time ($x > n$); or c) the correct number of marbles are passed out from an urn each time ($kx \geq n, x \leq n$), but sets of marbles of size x each can never make up a set of marbles of size n ($gcd(x, n) \neq x$). □

Assuming that a maximum of n marbles are to be selected from each urn, the next theorem shows the average number of marbles that can be selected before distribution begins.

Theorem 2: Assuming that the $m \cdot n$ marbles are uniformly distributed in the m urns, and a maximum of n marbles can be selected from each urn, the asymptotic expected number of marbles selected from each urn as $m \to \infty$ is $[1 - (e^{-n}n^n/n!)] \cdot n$.

Proof: Since the $m \cdot n$ marbles are distributed randomly into m urns, the number of marbles in any urn satisfies the binomial distribution with

$$\Pr(j) = \binom{m \cdot n}{j} \left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{mn-j} \tag{1}$$

where $\Pr(j)$ is the probability that j marbles are distributed into a particular urn. Since a maximum of n marbles can be selected from each urn, the expected number of marbles obtainable from each urn is

$$\begin{aligned} E_0(m, n) &= \sum_{j=1}^{n} j \Pr(j) + \sum_{j=n+1}^{mn} n \Pr(j) \\ &= n - n\left(1 - \frac{1}{m}\right)^{mn} - \sum_{j=1}^{n} (n - j) \Pr(j). \end{aligned} \tag{2}$$

The third term on the right-hand side of (2) has no closed form, but can be evaluated quite easily when n is small or in the limiting case when $m \to \infty$. Using the property that $\lim_{m\to\infty}(1 - (1/m)^m) = e^{-1}$, and that the binomial probability distribution becomes Poisson when m is large, we have

$$\begin{aligned} \lim_{m\to\infty} E_0(m, n) &= n - ne^{-n} - \sum_{j=1}^{n} (n - j) \lim_{m\to\infty} \Pr(j) \\ &= n - ne^{-n} - \sum_{j=1}^{n} (n - j) \frac{n^j}{j!} e^{-n} \\ &= n - ne^{-n} - ne^{-n}\left[\sum_{j=1}^{n} \frac{n^j}{j!} - \sum_{j=0}^{n-1} \frac{n^j}{j!}\right]. \end{aligned}$$

Simplifying,

$$\lim_{m\to\infty} E_0(m, n) = \left[1 - \frac{e^{-n}n^n}{n!}\right] \cdot n. \tag{3}$$

The asymptotic expected number of marbles selected from each urn, without any shift, is therefore $[1 - (e^{-n}n^n/n!)] \cdot n$. □

From (3), we see that as n is large, $\lim_{m\to\infty} E_0(m, n) \simeq n$ which means that most of the n marbles can be selected.

Continuing in a similar fashion, we can evaluate the fraction after one shift. In this case, a large combination of cases can occur. For example, in order to obtain j marbles in urn i after one shift, it depends on the number of marbles in urns $[(i - 1) \bmod m], [(i - 2) \bmod m], \cdots, [(i - k) \bmod m]$. Therefore, the probability distribution is a complicated sum of conditional probabilities. Similarly, the probability distribution after $s > 1$ shifts depends on the probability distribu-

tion after $s - 1$ shifts. The solution becomes iterative and it is unlikely that closed form solutions can be obtained. For our investigation, the following property is more important.

Theorem 3: Given an initial distribution of $m \cdot n$ marbles in m urns, and assuming that a maximum of n marbles can be selected from each urn, the total number of marbles selectable from the urns is a nondecreasing function of shifts.

Proof: Assume that the marbles in each urn are indexed with the topmost marble having the lowest index of 1. The index is an indicator of the marble's position in the urn. By adopting the convention that marbles received from near neighbors are given lower index numbers, we show that the index of a marble after a shift can never increase.

Consider a marble with index p in urn i. After a shift, this marble will be sent to urn $\ell = [(i + \lceil p/x \rceil) \bmod m]$ if $p \leq kx$; otherwise, it will stay in urn i. If the marble is sent to urn ℓ, the maximum index there is also p because the maximum number of marbles that urn ℓ has received with positions less than p must be coming from urns $[(i + 1) \bmod m], \cdots, [(\ell - 1) \bmod m]$, and this number must be less than p. If the marble stays in urn i, the maximum index can never be greater than p because urn i has sent out kx marbles and can receive a maximum of kx marbles from neighboring urns. Since the index of a marble after a shift can never increase, and marbles selectable before the shift remain selectable after the shift, the theorem is proved. □

Corollary 1: The average fraction of the $m \cdot n$ marbles selectable from the m urns is between 0.63 and 1.

Proof: From (2), it can be proved that $E_0(m, n)$ is an increasing function of n for a constant $m \cdot n$. Therefore, the minimum is obtained when $n = 1$. We have $\min[E_0(m, n)] = 1 - (1 - 1/(m \cdot n))^{m \cdot n}$. $\min[E_0(m, n)]$ decreases for increasing $m \cdot n$. As $m \cdot n \to \infty$, $\min[E_0(m, n)] \to 1 - 1/e > 0.63$. This is the minimum because the expected number of marbles selectable from each urn is a nondecreasing function of shifts (Theorem 3). For a given value of n (with $m \cdot n$ fixed), the average fraction is thus between 0.63 and 1. □

In order to choose the best single-stage interconnection configuration for the urns, we have to find the worst number of shifts in order to achieve a complete distribution. The following theorem proves the lower bound on the worst number of shifts in order to obtain a complete distribution.

Theorem 4: The maximum number of shifts to achieve a complete distribution in a system with m urns is greater than or equal to $m - 1$ for all combinations of n, k, and x that satisfy Theorem 1.

Proof: Since we are trying to find the lower bound of a maximum value, it is sufficient to show examples that would achieve this lower bound.

Suppose $x = n$ and $k = 1$. An initial distribution with all the marbles in one urn would require $m - 1$ shifts for a complete distribution. Assume that the shifts are done in a right cyclic fashion, the number of urns that contain x marbles each would be increased by one after a shift. Initially, there are $m - 1$ empty urns. Therefore, it requires $m - 1$ shifts in order to achieve a complete distribution.

Suppose $x = n$ and $k > 1$. The initial distribution is as follows: urn 0 has $2x$ marbles; urns $1, \cdots, m - 2$ have x marbles each; and urn $m - 1$ is empty. The "distance" in terms of the number of urns between the urn with 0 marble and the urn with $2x$ marbles is $m - 1$ initially. This distance is reduced by one after a right cyclic shift. It can be shown by a simple construction, and the proof is not shown here. The number of shifts required for a complete distribution is $m - 1$.

There are two other cases that have to be considered. They are $x < n, kx = n$, and $x < n, kx > n$. Examples to demonstrate the lower bound for these cases can be constructed similarly. However, more complicated notations have to be defined in order to prove the lower bound. They are omitted here for simplicity. Due to the existence of these examples, the theorem is proved. □

A stronger result than Theorem 4 is that the lower and upper bounds are equal to $m - 1$ and are independent of n, k, and x. However, the upper bound is not needed in the following discussion. Moreover, in order to prove this result, the way that the marbles are shifted in the urns must be understood. The proof is extensive and will not be shown here.

The $m - 1$ shifts required to achieve a complete distribution is much worse than the delay of multistage selection networks. However, single-stage networks have the advantage that relatively few shifts are required to obtain a large fraction of the set to be selected (Corollary 1). Further, the maximum number of shifts applies to all values of k and x. Therefore, the least expensive interconnection configuration, namely, $k = 1$, $x = n$, should be chosen.

In the remaining part of this section, some simulation results are shown. In Fig. 7, the efficiency of selection is plotted against the cost of different configurations with $k = 1$. The total number of marbles is 128. The cost is measured by the maximum number of marbles shifted from each urn. For example, for $m = 2, n = 64$, the maximum number of marbles shifted from each urn after one shift is 64. This cost measure is proportional to time when the marbles from each urn are shifted sequentially to its neighbor. Of course, when marbles are shifted in parallel, configurations with smaller m have better performance. Other simulation results (not shown) show that configurations with a larger degree of connections (k) and the same number of marbles have essentially the same performance using our cost measure and assuming that marbles from each urn are sent in parallel to the k neighbors. However, configurations with larger k are potentially more expensive.

In conclusion, for a given number of marbles, the number of urns should be kept as small as possible while the degree of connection should be kept at 1. We have neglected the overhead of inserting marbles into the urns. This will be considered in the next section.

VI. Design Considerations of MANIP

In this section, we analyze the effectiveness of MANIP and propose some guidelines for design. Assuming that all the lower bounds are distinct, the following parameters are first defined.

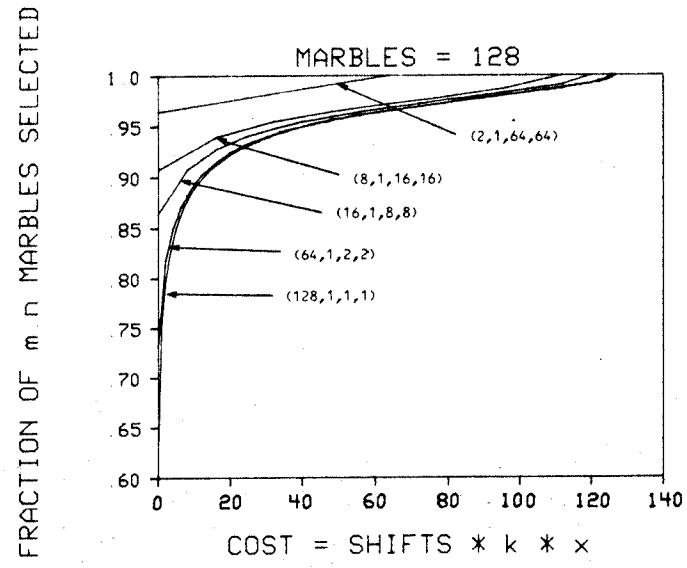


Fig. 7. A plot of the efficiency of single-stage selection versus the maximum number of marbles shifted from each urn. (Each configuration is a quadtuple (m, k, n, x).)

$\bar{T}_x$ = expected time to transfer x subproblems from processor i to processor j;
$= \begin{cases} x\bar{T}_1 & \text{serial transmission} \\ \bar{T}_1 & \text{parallel transmission} \end{cases}$

$\bar{S}_x$ = expected time to insert (delete) x subproblems into (from) the local list in a memory controller;
$= x\bar{S}_1$ (assuming no parallelism at each memory controller);

$\bar{E}_x$ = expected time to expand x subproblems into multiple subproblems and to find the corresponding lower bounds;
$= \begin{cases} x\bar{E}_1 & \text{sequential execution} \\ \bar{E}_1 & \text{parallel execution using } x \text{ processors} \end{cases}$

I_S = set of subproblems that are evaluated in a serial branch-and-bound algorithm under a best-first search

I_S = cardinality of $I_S = |I_S|$

h = number of shifts per iteration

i = number of memory controllers

j = number of processors per controller

$I_p(h,i,j)$ = number of iterations for a parallel branch-and-bound algorithm

$N_k(h,i,j) = i \cdot j \cdot R_k(h,i,j)$ = number of subproblems that belong to I_S in the kth iteration of the parallel branch-and-bound algorithm (this represents the number of "useful" subproblems expanded by the $i \cdot j$ processors), $0 \le R_k(h,i,j) \le 1$

$\tau_k(h,i,j)$ = time for executing the kth iteration of the parallel branch-and-bound algorithm

$T(h,i,j)$ = total time for executing the parallel branch-and-bound algorithm.

The following relationship is true:

$$I_S = \sum_{k=1}^{I_p(h,i,j)} N_k(h,i,j) = i \cdot j \cdot \sum_{k=1}^{I_p(h,i,j)} R_k(h,i,j).$$

Assuming that all the N_k's are independent and I_p is a stoppin time for the process, we can apply Wald's equation [17]

$$\bar{I}_S = \bar{I}_p(h,i,j) \cdot \bar{N}(h,i,j) = i \cdot j \cdot \bar{I}_p(h,i,j) \cdot \bar{R}(h,i,j) \quad (4$$

where a horizontal bar above a symbol indicates its expecte value. Similarly,

$$\bar{T}(h,i,j) = \bar{I}_p(h,i,j) \cdot \bar{\tau}(h,i,j). \quad (5$$

The optimal interconnection network for MANIP tha minimizes the expected completion time is given by th following theorem.

Theorem 5: Given m memory controllers and n processor per controller and assuming that the $m \cdot n$ subproblems wit the minimum lower bounds are distributed randomly, in orde to evaluate a parallel branch-and-bound algorithm in th minimum expected completion time, no shifting is necessar and the controllers do not have to be connected.

Proof: Assuming that a complete distribution is done i each iteration, the average number of iterations required i $\bar{I}_p(m-1,m,n) = \bar{I}_S/\bar{N}(m-1,m,n)$. The expected com pletion time of the algorithm using multistage and single stage selection networks are $O[\bar{I}_p(m-1,m,n)(\lfloor \log_2 m \rfloor \lceil \log_2 \log_2(m \cdot n) \rceil \bar{T}_1 + \bar{E}_n)]$ and $O[\bar{I}_p(m-1,m,n)((m-1)\bar{T}_n + \bar{E}_n)]$, respectively. When no shifting is done, we se from Corollary 1 that a minimum average of 63 percent o the subproblems are correctly selected assuming that the sub problems are uniformly distributed in the controllers. Thi implies that

$$\bar{N}(0,m,n) \ge 0.63\,\bar{N}(m-1,m,n)$$

or

$$\bar{I}_p(0,m,n) = \frac{\bar{I}_S}{\bar{N}(0,m,n)} \le \frac{\bar{I}_p(m-1,m,n)}{0.63}.$$

The expected completion time in the case of no shifting is les than or equal to $O[\bar{I}_p(m-1,m,n)\bar{E}_n/0.63]$. From this, i shows that a system with no shifting is the best.

The above theorem assumes uniform distribution of th subproblems in each iteration. However, the distribution be comes nonuniform as execution progresses. Therefore, on or more shifts may be needed in each iteration in order t randomize the distribution. Furthermore, a network is neces sary for balancing the load in the system. We conclude tha a unidirectional ring network is the minimal interconnectio network for effective execution of parallel branch-and-boun algorithms.

When using a unidirectional ring network ($k = 1$), x, th number of subproblems sent in each shift to a neighborin processor must be n (Theorem 1).

In most applications, the performance of using one shif per iteration is more than adequate. Since the number o shifts is small, shifts can be carried out in overlap with sub problem expansions. Further, although a processor may not b expanding one of the subproblems with the minimum lowe bounds, it may be expanding a subproblem that will be evalu ated in a future iteration. This contributes to a higher degre of effectiveness.

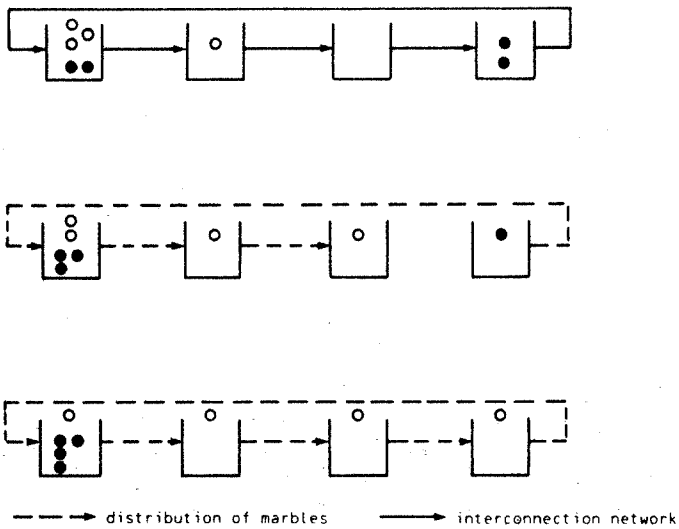


Fig. 8. Decrease in urns containing black marbles when distribution is complete ($m = 4, n = 1, k = 1, x = 1$).

It is interesting to note that when a large number of shifts are made, the distribution of the first global minima improves; that is, the number of urns containing white marbles increases. However, the distribution of the $(mn + 1)$st to $2mn$th global minima, which are represented as black marbles, may be worse. The phenomenon is illustrated in Fig. 8. This is due to the fact that an urn of white marbles can block the distribution of black marbles. Black marbles from neighboring urns may flow into an urn of white marbles. Since the white marbles are distributed first, black marbles will accumulate in this urn. This distribution is important because it governs the distribution of the white marbles in the next iteration (when the black marbles in this iteration become the white marbles in the next iteration). The average number of urns containing black marbles after a complete distribution is actually smaller than the average number of urns if the marbles were distributed randomly. Fortunately, the change in distribution is insignificant when the number of shifts is small.

Using the optimal interconnection, that is, $k = 1, x = n$, the last questions to be answered are how to choose n, the number of processors for each memory controller, and m, the number of memory controllers. When h shifts are performed in each iteration, the average total time in executing the algorithm is [according to (5)]

$$\overline{T}(h, m, n) = \frac{\overline{I}_S}{m \cdot n \cdot \overline{R}(h, m, n)} \cdot [h(\overline{T}_n + \overline{S}_n) + \overline{S}_n + \overline{E}_n + \overline{S}_{2n}]. \quad (6)$$

The terms on the right-hand side of (6) correspond to the overhead in each iteration of shifting and inserting subproblems into the local list, deleting from the local list, expanding and evaluating the lower bounds of subproblems, and inserting new subproblems back into the local list. A simplified assumption is made that each subproblem is always expanded into two new subproblems.

Since the overhead of insertion and deletion from the local list is very small, the terms $\overline{S}_n$ and $\overline{S}_{2n}$ can be neglected. For fixed values of h and m, the ratio

$$\left.\frac{\overline{T}(h, m, n_1)}{\overline{T}(h, m, n_2)}\right|_{h,m} = \frac{n_2 \overline{R}(h, m, n_2)(h\overline{T}_{n_1} + \overline{E}_{n_1})}{n_1 \overline{R}(h, m, n_1)(h\overline{T}_{n_2} + \overline{E}_{n_2})} \quad (7)$$

measures the merits of two different degrees of multiprocessing in each memory controller. If shifts and evaluations are done sequentially ($\overline{T}_n = n\overline{T}_1, \overline{E}_n = n\overline{E}_1$), we have

$$\left.\frac{\overline{T}(h, m, n_1)}{\overline{T}(h, m, n_2)}\right|_{h,m} = \frac{\overline{R}(h, m, n_2)}{\overline{R}(h, m, n_1)}. \quad (8)$$

From the analysis of the urn model (3) and from simulations using vertex-covering problems, we found that $\overline{R}(h, m, n_1) \geq \overline{R}(h, m, n_2)$ for $n_1 > n_2$. This implies that n should be set as large as possible.

Likewise, when shifts and evaluations are done in parallel ($\overline{T}_n = \overline{T}_1, \overline{E}_n = \overline{E}_1$), we have

$$\left.\frac{\overline{T}(h, m, n_1)}{\overline{T}(h, m, n_2)}\right|_{h,m} = \frac{n_2 \overline{R}(h, m, n_2)}{n_1 \overline{R}(h, m, n_1)}. \quad (9)$$

Again, n should be set as large as possible.

In practice, when n is chosen above a maximum value, the total execution time is not going to improve. This is due to the fact that the set of "useful" subproblems at the head of the subproblem list is finite and limited. Once n exceeds the size of this set, the additional evaluations will be wasted. This maximum value has been evaluated in [28]. In general, n should be set as large as possible without exceeding this maximum.

For fixed values of h and n, the ratio

$$\left.\frac{\overline{T}(h, m_1, n)}{\overline{T}(h, m_2, n)}\right|_{h,n} = \frac{m_2 \overline{R}(h, m_2, n)}{m_1 \overline{R}(h, m_1, n)} \quad (10)$$

measures the merits of two different numbers of memory controllers. From the analysis of the urn model and the simulation results, $\overline{R}(h, m_1, n) \leq \overline{R}(h, m_2, n)$ for $m_1 > m_2$, but normally, the rate of increase of $\overline{R}(h, m_2, n)/\overline{R}(h, m_1, n)$ is much smaller than the rate of decrease of m_2/m_1. This implies that m should be set as large as possible.

The final question to be answered is what should be the values of m and n when $m \cdot n$ is constant. In this case, there are a fixed number of processors implemented in hardware, and we have to determine the number of memory controllers that results in the maximum efficiency. Since $E_n = E_1$, we have

$$\left.\frac{\overline{T}(h, m_1, n_1)}{\overline{T}(h, m_2, n_2)}\right|_{h,mn} = \frac{\overline{R}(h, m_2, n_2)(h\overline{T}_{n_1} + \overline{E}_1)}{\overline{R}(h, m_1, n_1)(h\overline{T}_{n_2} + \overline{E}_1)}. \quad (11)$$

Regardless of whether the subproblems are shifted sequentially or in parallel, the major overhead is in the shifts except in the case of $m = 1$ which has no shifting involved. Further, $\overline{R}(h, 1, mn) \geq \overline{R}(h, m, n)$. It is obvious in this case that m should be set to 1, that is, the processors are centrally controlled by a single controller.

The single controller may become the bottleneck of the system. If more than one controller is used, m should be set as small as possible if multiple subproblems are shifted in

parallel. However, if shifts are done serially, the major overhead is in the shifts. Therefore, m should be set as large as possible in order to result in a small n (for constant $m \cdot n$). All this can be deduced from (11).

VII. Simulation Results

Following are some results on the performance of the parallel branch-and-bound algorithm for the vertex-covering problem. It is assumed in the simulation model that the controllers are connected by a unidirectional ring network. All the controllers and processors operate synchronously. Each processor picks up a subproblem for expansion from the local list. Newly created subproblems are inserted back, and a number of shifts are performed before the next cycle starts. The two sequential steps just described are an *iteration*. Because it is more important to study the algorithm's feasibility at this time, the details of an implementation were not simulated, and only the number of iterations required were compared.

The simulation model was written in *C* language and implemented on a DEC VAX 11/780 computer with virtual memory. The simulation program is complex because complicated data structures and heap sort are used to achieve a fast insertion and deletion time. The VAX 11/780 was chosen because of the availability of virtual memory. For example, if each memory controller has a storage capacity of 300 subproblems of 340 bytes each (for a typical vertex-covering problem), the total memory required for 64 controllers is about 64 Mbytes. The capacity of 300 subproblems in each controller is still insufficient for some cases. When memory space is exhausted, the best-first search is switched to a depth-first search. For simplicity, the algorithm does not switch back to a best-first search when memory space is available again.

The following parameters are used: NODE, the number of nodes in the graph; PROB, the probability that an edge exists between two vertices in the graph; SHIFT, the number of shifts per iteration (n subproblems are shifted each time); and BFSMEM, the maximum number of subproblems that can be stored in a memory controller for a best-first search. The number of nodes in the graph varied between 10 and 80. Parameters PROB and m were chosen from the sets $\{0.1, 0.25\}$ and $\{1, 2, 4, 8, 16, 32, 64\}$, respectively. Ten graphs were generated randomly for each combination of NODE, PROB, SHIFT, m, and n. Simulations took about 100 h of computer time on the VAX computer.

A selected set of the simulation results are shown in Figs. 9–12. Other simulation results have been shown in an earlier paper [20]. Fig. 9 shows the average number of iterations for a depth-first search alone. It is observed that the number of iterations increases exponentially as the problem size increases. Further, the spacing between adjacent curves is almost constant which indicates that the usual effects of diminishing return of parallel processing did not occur here. In Fig. 10, we have investigated the effects of varying m and n for a constant $m \cdot n$. It is seen that there is very little difference between different combinations of m and n.

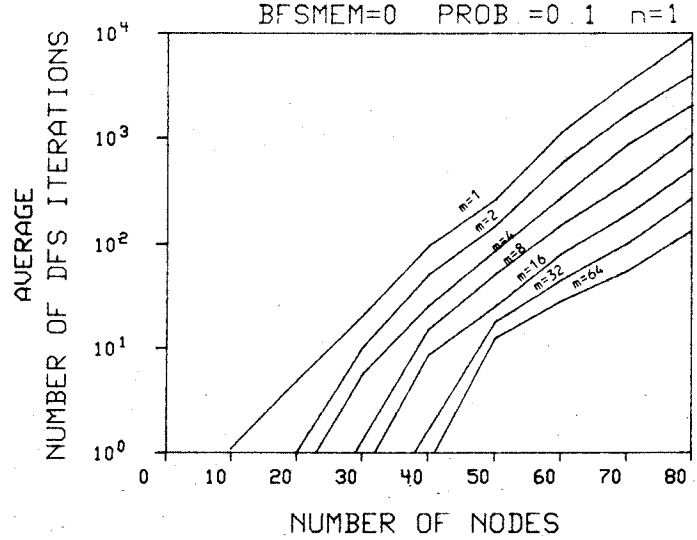


Fig. 9. Performance of depth-first search for different number of processors.

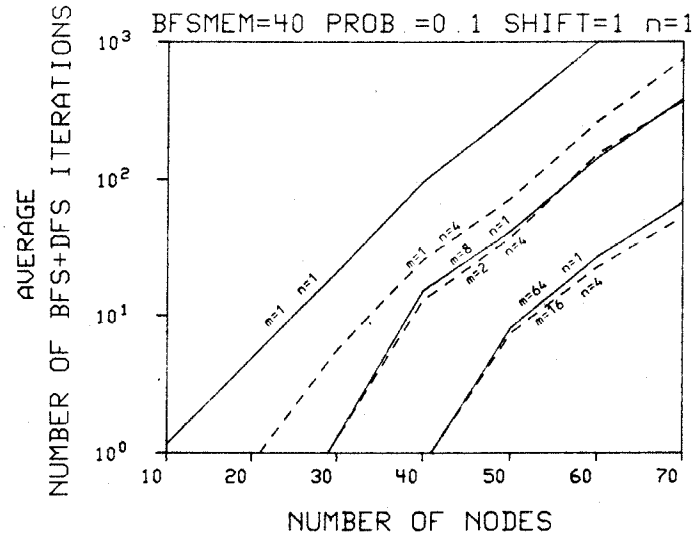


Fig. 10. Performance of best-first and depth-first searches combined for different combinations of m and n.

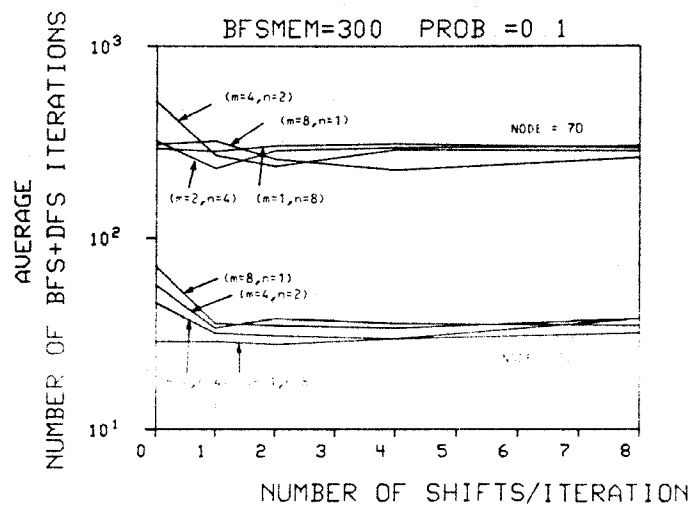


Fig. 11. Performance of best-first and depth-first searches combined with $m \cdot n = 8$.

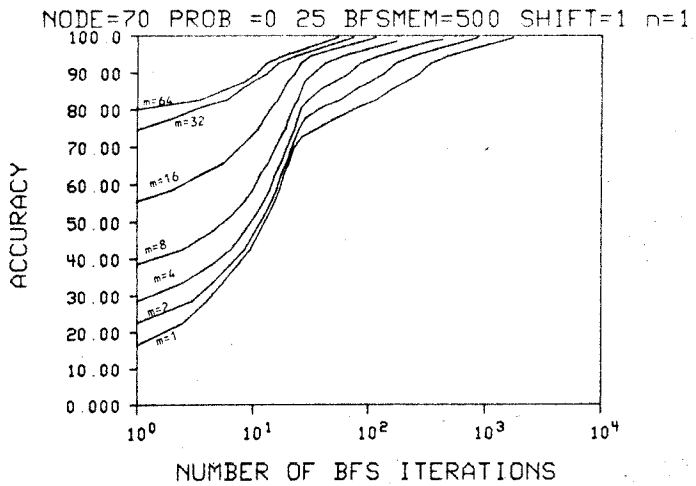


Fig. 12. The reduction in the number of iteration as accuracy decreases.

Comparing Figs. 9 and 10, it is seen that the number of iterations is reduced slightly by allowing BFSMEM = 40. However, in order for a best-first search to dominate the performance, BFSMEM has to be set to at least 300 in this case.

In Fig. 11, we show the effects of shifting on performance. We assume that for SHIFT = 0, an empty memory controller can request subproblems from neighboring controllers, thereby balancing the load in the system. Without this mechanism, no shifting results in very poor performance. There is basically very little improvement in performance for a large number of shifts per iteration. This observation agrees with our previous result that the ring network is very important in balancing the load and randomizing the distribution of subproblems among the controllers. It must be pointed out that sorting is still necessary in the controllers.

The last graph (Fig. 12) shows the merits of allowing approximate solutions. BFSMEM is set to be sufficiently large so that the algorithm does not have to switch to a depth-first search. It is seen that the accuracy is exponential with respect to the number of iterations which agrees with our previous results [22]. Furthermore, the use of parallel processors has a significant impact on improving the accuracy.

In the performance results shown, we observe that there are knees in the curves (Figs. 9, 10, 12). The reason for these is that the total number of iterations required in those cases to the left of the knees is very small and approaches the number of processors. In all cases, there is always a minimum overhead in starting the branch-and-bound process which is the minimum path length from the root to a leaf. Further increase in the number of processors for small problems results in underutilization of processors and degradation in efficiency.

In summary, the algorithm performs well when memory is sufficient to perform a best-first search. Once it is switched to a depth-first search, the average number of iterations increases rapidly. Furthermore, these results indicate that the improvement in using n processors is very close to n, and the effects of diminishing return is very small for the graphs tested.

VIII. Conclusion

In this paper, we have proposed and studied the design of MANIP, a parallel computer architecture for processing NP-hard problems. The most general algorithm that can be used to solve a wide variety of NP-hard problems on a uniprocessor system, optimally or suboptimally, is the branch-and-bound algorithm. A parallel version of the algorithm was studied in this paper.

To minimize the execution time, evaluation of subproblems with the minimum lower bounds is essential. This requires a combination of sorting and merging. We have investigated several alternatives for maintaining the list of subproblems so that subproblems with the minimum lower bounds can be selected. We have found that a unidirectional ring network is cost effective in selecting subproblems to evaluate and balancing the load in the system. The proposed ring network is reliable because it is simple and can be reconfigured. Faulty processors can be switched off the network without affecting the performance of other processors. Redundant rings can be used to increase the reliability of the network.

To maintain a large list of subproblems in each memory controller, we have proposed a system in which subproblems with large lower bounds are "paged" out onto the secondary storage. It was found that a hundred times reduction in the paging traffic can be attained as compared to a virtual memory with LRU replacement algorithm [30].

Another important implication from this research is that approximate branch-and-bound algorithms combined with parallel processing is very effective in reducing the exponential nature of the evaluations. Theoretical properties of approximate branch-and-bound algorithms have shown that a linear reduction in accuracy results in an exponential reduction in the number of iterations [22]. Theoretical considerations have shown that anomalous behavior due to parallelism can be avoided in most cases [28], [30]. Parallel processing with virtual memory and approximation are, therefore, a general methodology for solving the intractable NP-hard problems.

References

[1] V. E. Alekseyev, "Sorting algorithms with minimum memory," *Kibernetika*, vol. 5, no. 5, pp. 99–103, 1969.
[2] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Computer Conf.*, vol. 32, Apr. 1968, pp. 307–314.
[3] B. C. Desai, "The BPU, A staged parallel processing system to solve the zero-one problem," in *Proc. ICS'78*, Taipei, Taiwan, Dec. 1978, pp. 802–817.
[4] O. I. El-Dessouki and W. H. Huen, "Distributed enumeration on network computers," *IEEE Trans. Comput.*, vol. C-29, pp. 818–825, Sept. 1980.
[5] T. Y. Feng, "A survey of interconnection networks," *Computer*, vol. 14, pp. 12–27, Dec. 1981.
[6] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-completeness*. San Fransico: Freeman, 1979.
[7] V. O. Groppen, "Efficiency of parallel computation for the solution of combinational extremum problems," *Avtomat. Telemekh.*, pp. 114–122, May 1980.
[8] M. Held and R. M. Karp, "The traveling salesman problem and minimum spanning trees," *Oper. Res.*, vol. 18, pp. 1138–1162, 1970.
[9] R. M. Karp, "A patching algorithm for the nonsymmetric traveling-salesman problem," *SIAM J. Comput.*, vol. 8, no. 4, pp. 561–573, Nov. 1979.

[10] H. T. Kung and C. Leiserson, "Highly concurrent systems," in C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.
[11] E. L. Lawler and D. W. Wood, "Branch-and-bound methods: A survey," *Oper. Res.*, vol. 14, pp. 699–719, 1966.
[12] D. D. Marshall, "A parallel processor approach for searching decision trees," in *Proc. 1977 Int. Conf. on Parallel Processing*, MI, 1977, pp. 199–201.
[13] T. A. Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *ACM Computing Surveys*, vol. 14, no. 4, pp. 533–552, Dec. 1982.
[14] L. Mitten, "Branch-and-bound methods: General formulation and properties," *Oper. Res.*, vol. 18, pp. 24–34, 1970.
[15] F. P. Preparata, "New parallel-sorting schemes," *IEEE Trans. Comput.*, vol. C-27, pp. 669–673, July 1978.
[16] C. V. Ramamoorthy, J. L. Turner, and B. W. Wah, "A design of a fast cellular associative memory for ordered retrieval," *IEEE Trans. Comput.*, vol. C-27, pp. 800–814, Sept. 1978.
[17] S. M. Ross, *Applied Probability Models with Optimization Applications*. San Francisco: Holden-Day, 1970.
[18] S. Sahni, "General techniques for combinational approximation," *Oper. Res.*, vol. 25, no. 6, pp. 920–936, 1977.
[19] L. G. Valiant, "Parallelism in comparison problems," *SIAM J. Comput.*, vol. 4, no. 3, pp. 348–355, Sept. 1975.
[20] B. W. Wah and Y. W. Ma, "MANIP—A parallel computer system for implementing branch-and-bound algorithms," in *Proc. 8th Annu. Symp. on Comput. Arch.*, 1982, pp. 239–262.
[21] B. W. Wah and K. L. Chen, "A partitioning approach to the design of selection networks," *IEEE Trans. Comput.*, vol. C-33, pp. 261–268, Mar. 1984.
[22] B. W. Wah and C. F. Yu, "Probabilistic modelling of branch-and-bound algorithms," in *Proc. COMPSAC*, Nov. 1982, pp. 647–653.
[23] A. C. C. Yao, "Bounds on selection networks," *SIAM J. Comput.*, vol. 9, no. 3, pp. 566–582, Aug. 1980.
[24] ——, "On parallel computation for the knapsack problem," *J. Ass. Comput. Mach.*, vol. 29, no. 3, pp. 898–903, July 1982.
[25] C. F. Yu, "Design of virtual memory operating system to support branch and-bound algorithms," M.S. thesis, Purdue Univ., W. Lafayette, IN Aug. 1983.
[26] M. Imai, T. Fukumura, and Y. Yoshida, "A parallelized branch and-bound algorithm: Implementation and efficiency," *Syst. Comput Controls*, vol. 10, no. 3, pp. 62–70, 1979.
[27] T. H. Lai and S. Sahni, "Anomalies in parallel branch-and-boun algorithms," in *Proc. 1983 Int. Conf. on Parallel Processing*, Aug. 1983 pp. 183–190.
[28] G. J. Li and B. W. Wah, "Computational efficiency of parallel approxi mate branch-and-bound algorithms," Purdue Univ., W. Lafayette, IN Tech. Rep. TR 84-6, Mar. 1984.
[29] D. R. Smith, "Random trees and the analysis of branch-and-boun procedures," *J. Ass. Comput. Mach.*, vol. 31, pp. 163–188, Jan. 1984
[30] B. W. Wah, G. J. Li, and C. F. Yu, "The status of MANIP—A multi computer for solving combinatorial extremum-search problems," i *Proc. 11th Annu. Int. Symp. on Comput. Arch.*, June 1984.

Benjamin W. Wah (S'74–M'79), for a photograph and biography, see p. 26 of the March issue of this TRANSACTIONS.

Y. W. Eva Ma (S'79–M'81) received the A.B. an M.S. degrees in computer science and the Ph.D degree in electrical engineering and computer sci ence from the University of California, Berkeley.

She is now an Assistant Professor in the De partment of Computer and Information Science University of Pennsylvania, Philadelphia, PA. He research interests include computer architecture an complexity theory.

Dr. Ma is a member of Association for Computin Machinery.